APPENDIX A

Memo: VERP Versus Message ID
Revision : 2.0

Chris Shenton, Manjusha Gadamsetty, Lee Mann, Gautam Yegnanarayan

June 15, 2000

1 Introduction

We have been using the phrase "message ID" and "VERP" somewhat interchangably, leading to some confusion on what each is. I would like to avoid "message ID", leaving it as a database table auto-generated ID used in the "message" table's primary key.

"VERP" is defined by DJB in http://cr.yp.to/proto/verp.txt as a way to identify bounced mail target recipients even from uncooperative remote mail transport agents. The *envelop sender* is set to a value unique to the intended recipient. A remote MTA is supposed to send bounced mail to the envelop sender (rather than anything in the message headers). When the bounce is received, the original MTA can examine the VERP to determine exactly what recipient failed.

Typically, the recipient's email address is encoded into the VERP since this is usually what is wanted, but it could be anything. We want to use the VERP to identify not just the target email address, but also a "recipient" (which isn't just an email address), its associated submission, customer, etc.

We plan to use this VERP not only for email envelop sender but also within email headers as the Errors-To address, as a component of click-through and invisible image tracking URLs, and other places where we want to track unique messages.

Becuase of this overload of functions, I would prefer to avoid the term "VERP" for this, but we'll need to come up with a term we can all agree to. Perhaps "TrackingID" or some such, but perhaps that's been used already; suggestions welcomed. Until we do come up with a new term, I'll use "VERP" in this paper.

2 Assumptions

2.1 One recipient per message, period

There has been much discussion about the possibility of sending a message which has multiple recipients within the single message connection; we reject this.

While it seems like it *might* improve efficiency, it precludes the features which are critical to eFoton.

1. Message content cannot be unique to each recipient
2. We can't track which recipient read the message via stealth images or click-throughs.
3. We can't offer recipient-unique web pages from the WAG
4. We can't gather intelligence about the recipient such as mail client, preferred read time, etc.

Since each message will have only one recipient, we can then use a unique identifier on each outbound message to track message delivery, receipt by recipient, and subsequent follow-through by the recipient.

2.2 Expect brain-damaged MTAs

The VERP must survive transport through brain-damaged MTAs, such as MicroSoft products which truncate values left of the @-sign to 32 characters. Specific suggestions for syntax are discussed below.

3 Variable Encoding Considered Harmful

We initially suggested encoding important variables such as the database MessageID and SubmissionID into the VERP. This section describes why we believe this is not a good idea, but boils down to scalability.

3.1 Variable transparency is bad

While it's easy to use something like the MessageID as the VERP, exposing this internal key datum to the world puts the system at risk. A remote user could send mail to the eFoton OOBHandler with the MessageID and it would be interpreted as a bounce; while bad, this only affects the miscreant user. With a transparent VERP they could alter the VERP value and simulate a bounce of someone *else's* mail — clearly unacceptable. A forged bounce would cause eFoton to roll-over and generate a new message for the "failed" recipient.

We have to protect the privacy of any data we use in the VERP. It should not be possible for the casual miscreant to to affect eFoton with forged email. Ideally, it would be impossible for outsiders to generate email with a forged VERP which would match a real eFoton identifier.

3.2 Scalability

After the initial MessageID proposal, we realized we wanted the SubmissionID available through the VERP as well; it would help us in database queries, possibly saving an additional step. While you could query the database on the MessageID to get the SubmissionID, this represents one extra query. Likewise the SubmissionID can be used to find the CustomerID, SubmissionChunkID, DAtype, and all manner of other data. Perhaps they should be included in the VERP as well. Clearly this is not scalable: there is insufficient room in the VERP to encode all the meta-data about a message. (Below we present some numbers to indicate how much data can be encoded).

3.3 VERP table

The solution is to implement a database table whose primary is the VERP. Other elements in the table would include all the data one would want to quickly identify from a VERP which enters the eFoton system, such as MessageID, SubmissionChunkID, SubmissionID, CustomerID, RecipientID, MessageExpirationDT.

To prevent data duplication, the other fields in the VERP table would be foreign keys which are indices into other tables. While the VERP table represents an "extra" database query before the system can do any work, it should provide sufficient information to assist subsequent queries, possibly saving later queries in the process. (I'm not sure how we would put the ExpirationDT in the table without actual duplication, unless we have an Expiration table as well).

In actuallity, the information in the proposed VERP table is already present in the Message table, so we'll just use that one.

4 What's in a VERP?

How is the VERP constructed? I mean: what's in it? We want to prevent revealing private information and we have limitations on the size of the VERP itself as well as what characters we can use in it.

Since the VERP is not related to the MessageID or any other datum, it can be any arbitrary value — so long as it's unique.

4.1 Opacity and uniqueness

As discussed above, we must conceal internal data (such as MessageID) to prevent outsiders hacking eFoton's behavior. We could generate random numbers, use hash functions, or timestamps but there are issues with each which must be addressed before implementation.

Using random numbers would require querying the VERP table to see if the proposed number was already used. This could be slow as the VERP table grows.

A hash of some database-generated unique value, such as a monotonically-increasing index value, would *almost* guarantee uniqueness. Cryptographically-strong hashes are designed with this property. We can modify the format of the hash to use the mathematical "base" appropriate to the VERP envelop, as explained below (e.g., instead of MD5's base-16 we could use base-40 if the alphabet has 40 characters). Curtis expressed concern about the speed of hashes such as "MD5". Even if the hash is sufficiently fast, are we comfortable with the collision risk level?

A time stamp could be used, but even with a sufficiently small resolution (e.g., DJB's "accustamp") it's possible multiple messages would be generated in the same single interval. We would need to append some sequence number to prevent this. Keeping track of the sequence may be tedious.

4.2 Length

The VERP is the local-part of the envelop sender address, that is, the stuff before the @-sign. While I can't find any maximum length for this in RFC-822, there are practical limits imposed by brain-damaged MTAs. It has been reported that at least one MicroSoft mail system truncates local-parts which are longer than 32 characters. Therefore, we will restrict the VERP length to be 32 characters.

4.3 Characters

For LatestEdition, Manjusha wrote:

```
I went through rfc822 document and came out with all valid characters
that can be present in the local-part of email addresses where :
local-part@blah.blah is an email address.

The local-part can contain :

0-9, a-z, A-Z, !, #,$,%,^,&,',*,+,-,/,=,?,_,`(Back tick), ~, {, |,}

Also a "." ( I mean, a period) can be present, but it should not be at
the beginning of local-part.
I mean, manju.gadams@uucom.com is valid.
But    .manju@uucom.com is not valid.
So, we should, from now on, be able to support and  take care of all
these characters.
```

RFC-822 defines the address syntax explicitly; below are the definitions relevant to the local-part:

```
addr-spec   =  local-part "@" domain        ; global address
local-part  =  word *("." word)             ; uninterpreted case-preserved
word        =  atom / quoted-string
atom        =  1*<any CHAR except specials, SPACE and CTLs>
quoted-string = <"> *(qtext/quoted-pair) <">; Regular qtext or quoted chars.
specials    =  "(" / ")" / "<" / ">" / "@"  ; Must be in quoted-
            /  "," / ";" / ":" / "\" / <">  ;  string, to use
            /  "." / "[" / "]"              ;  within a word.
qtext       =  <any CHAR excepting <">,     ; => may be folded
               "\" & CR, and including
               linear-white-space>

This reserved local-part must be matched without sensitivity to
alphabetic case, so that "POSTMASTER", "postmaster", and even
"poStmASteR" is to be accepted.
```

Do all MTAs preserve characters according the the RFC? If not, it would be safest to err on the side of conservatism. Unless we are certain that letter case is preserved, we should restrict ourselves to lower-case. Old mailers treated ! and % as mail-path separators, and the pipe operator — is frequently rejected for security reasons. Are there other characters which should be eliminated? Double-quotes must appear in matched pairs, so we'll omit them. I'm suspiscious that mailers will mangle some of those characters like the single quotes (forward and backward), curley braces, and shell meta-characters. Let's play it safe and just use characters which are known to be commonly supported, and include the "." (dot) and "-" (dash) with the restriction that they may not be used as the first character, per the RFC. So our VERP alphabet is comprised of:

```
0-9 a-z _+.-
```

This does leave us with enough room for the information we want to encode, as the math below explains.

4.4 How many?

The number of possible VERPs is therefore 40^{32} or about 1.84×10^{51} different values. If we assume 10^6 messages/hour, or about 9×10^9 messages/year, this means we'll run out of unique VERPs after 2.05×10^{41} years. This seems like plenty. :-)

Perhaps the VERP *does* have "enough room" to encode a variety of different values such as various IDs. For example, if we have 10 variables we want to encode, and each can have a value up to 9e9 (say) then this would require about 3.5×10^{99} unique VERPs. We'd run out very quickly.

While the bit-twiddlers might argue that we won't need a billion values for CustomerID, or even SubmissionID, and maybe we won't want to encode as many as 10 different variables, we'd still only be able to reduce it by orders of magnitude. It still won't scale if we decide down the road that it would be really helpful to have some other kind of ID encoded. Using a VERP table will allow us to add anything we find convenient.

5 Compromise: Encode VERP ID and Critical Data

On 2000/05/15 Jerry, Manjusha and Chris discussed trade-offs of trying to cram a bunch of vital variables into the VERP, putting only the VERP table VerpID primary key, or some combination. The size of the data structure we have to work with is large for a single datum, but rather restrictive if we want to encode multiple possibly-large values.

On 2000/05/24 Chris corrected the VERP size from 30 to 32, which gives us a slightly larger size, but decreased the alphabet from 52 characters to the most conservative 40, which greatly reduces the space.

5.1 Triage most critical information

We realized that although MessageId, RecipientId and such were important, the first thing ever done with a VERP is to check the Expiration data and time. If the expiration time has passed, a simple response is required and future database hits can be eliminated. Therefore encoding the Expiration into the VERP can save us over 50% of our database hits associated with processing OOB bounces, invisible read-tracking images, and click-through URLs.

Obviously, we want to encode a VERP ID which will be used as a key into the Message table which holds foreign keys to the other important tables of interest. This ID must be unique for all time so it is expected to have potentially large values, requiring relatively large storage.

We also want some bits set aside to store a VERP-encoding version, so if we ever decide to change the contents the VERP processors can distinguish formats. Vital, but not very big.

We'll want some kind of checksum on the data to detect tampering and forgery by remote users. A simple checksum would be the one-bit parity on a data byte; it can't detect two-bit errors and can be easily spoofed by an attacker. Strong cryptographic hashes include the MD5 algorithm which has a hash represented by 32 hexadecimal digits; unfortunately our VERP size is too small to encode that. We'll need some compromise here.

Finally, we'll want some world-visible symbol at the front of the VERP to help our VERP processors (especially the OOB Handler looking at randomly-formatted bounce messages) locate VERPs within messages. For example, the SCCS system uses the 4-character string "@(#)" to help it locate SCCS identifiers; the idea is that this string is unlikely to occur in programs and other text. We can use something similar which is unlikely to occur in customer-generated mail, bounces, but fits within the alphabet allowed in a VERP. Note that this must *not* be encoded like the other data since we need to be able to find it easily in plain-text bounce messages.

5.2 Component Sizing

If we examine the elements we must cram into the VERP in a orderly fashion we can estimate needed sizes and determine that the attributes above will all fit. We'll align each element on VERP character boundaries for ease of access.

5.2.1 Signifier

We'll use a 4-character Signifier selected from our 40-character alphabet, leaving us with 28 characters for the rest of the VERP.

Recall we cannot use "-" or "." for the first character of our VERP, so we'll make the Signifier the first part and avoid use of these two characters in the first position. We will chose the Signifier to be "e+f0" (ee plus eff zero): starts with an alpha, the plus is unusual in the middle of a username, and the trailing zero will merge with the following VERP ID which may well start with zeros.

5.2.2 VERP ID

If we assume we'll begin with one million messages delivered messages per hour the first year, I think we should consider that we may quickly rise to around ten million message delivery attempts per hour (failed deliveries and re-attempts, increasing demand and increased capacity). If we expect we'll need this many unique IDs each year for ten years, we'll need $10 \times 10^6 \times 24 \times 365 \times 10 = 8.76 \times 10^{11}$ unique VERP IDs. We can accommodate this with 8-characters in our alphabet because $40^8 = 6.55 \times 10^{12}$; we're left with 20 characters.

5.2.3 Expiration

For now we'll settle on using the UNIX-style date/time which encodes the number of seconds since "the epoch" (1970/01/01 00:00:00 UTC) in a 4 byte value. This requires $2^{32} = 4.29 \times 10^9$ values. We can fit this into 7-chars in our alphabet because $40^7 = 1.64 \times 10^{11}$. We have 13 remaining.

(Note that this will run out in 2038/01/18; if we care we will have to use some other time standard, such as TAI, but these will most likely require more precious bits).

5.2.4 Checksum

We'll arbitrarily decide on 4 bytes (64 bits) for the checksum, which consumes $2^{64} = 1.84 \times 10^{19}$ bits and another 7 characters, leaving 6 characters free.

Note that the checksum should not include the Signifier nor Version in its calculation but must include the VERP ID, Expiration, and Unused (random) values since we want to detect tampering on any of them. Including the unused/random data in the hash will allow us to use the same hash function in the future if we decide to put data there.

The hash should be computed on the three combined fields as represented by the base-40 encoded text of the VERP text. This will allow easy parsing of the VERP string, and will detect corruption before any attempt is made to decode the variables represented by the VERP text.

The algorithm is left as an exercise to the implementer. Check Bruce Schneiers's *Applied Cryptography* for suggestions.

5.2.5 Unused is random

Any unused space remaining in the VERP should be set to a random value before encoding so that the encoded data will not look like a suspicious string of zeros; the checksums will be calculated including this data to detect tampering.

5.2.6 Version

We'll reserve one of our characters for the VERP version, giving 40 possible versions. Now we're down to 27 characters for the rest of the VERP. Putting the Version character at the end of the VERP text string allows us to avoid ending the userid portion with something which might end in an invalid final email address character such as "." or "-". The version, like all other numbers, is encoded with the characters from our base-40 alphabet, starting with "0".

Note that if we ever change the format of the VERP text or the interpretation of it (e.g., put something useful in the Unused area) then we *must* increment the Version so that software can decode it properly and other modules can interpret the values correctly.

5.2.7 Does it fit?

Here is a graphical breakdown of the 32-characters in the VERP and which positions are used for what values.

```
00000000011111111112222222222333
12345678901234567890123456789012 Position

SSSSVVVVVVVVEEEEEEErrrrrCCCCCCC# Use

UseCodes:
S signifier tag (e+f0)
V VERP ID
E expiration date (4 bytes before encoding)
C checksum of VEr data (4 bytes before encoding)
r random, unused
# version of verp format in use
```

5.3 Encoding/Decoding

As alluded to in the above, we'd want a pair of verp_encode() and verp_decode() functions to do all this work for us. We're using character boundaries for each field to simplify the encoding and speed the calculations. Of course all values would have to use our base-40 math to encode the values in the alphabet, using an alphabet which is whose numeric order is exactly as follows:

```
0123456789abcdefghijklmnopqrstuvwxyz_+.-
```

5.4 Issues

If the format of the VERP is made public, miscreants can forge VERPs with proper checksums. I've never believed in "security through obscurity" and I know darn well the crypto community has simple solutions for this. Any ideas? Ideally, we could publish the format and still be immune from attack.

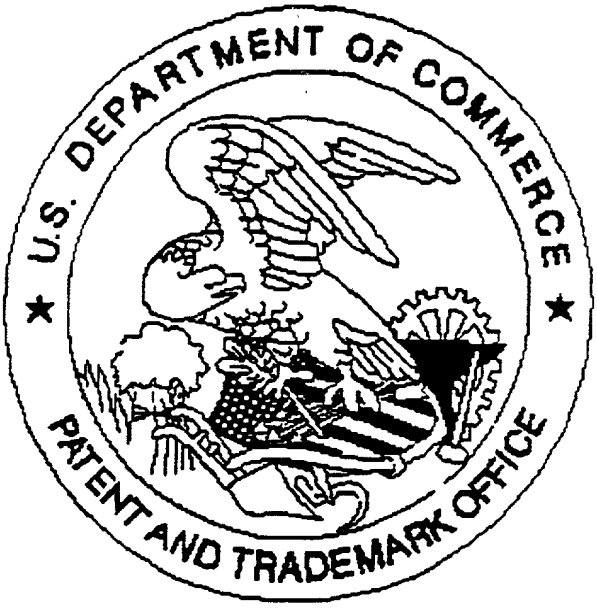
Can we find small yet cryptographically-sound hashes? Do we need the entire 4-bytes for the digest of such a small "message"?

Some of the VERP space is unused; should we encode an 8-byte TAI date/timestamp? A larger checksum? More room for larger VERP IDs?

Can we use the Message.message_id as the value of the Message.verp_id? If we ever decide to make message_id's non-global (e.g., start at zero with each new submission, separating the namespace by submission number) then we cannot use this.

United States Patent & Trademark Office

Office of Initial Patent Examination -- Scanning Division



Application deficiencies found during scanning:

☐ Page(s)_____of_____ were not present for scanning. (Document title)

☐ Page(s)_____of_____were not present for scanning. (Document title)

☑ ***Scanned copy is best available.*** Drawings